Page 39, line 45-46.

APPENDIX A
Docket No. SF/0014.01

# TrueSync Developer's Guide

## *Contents*

Section 1: Developer's Overview of the TrueSync Architecture
Section 2: Lifecycle of a Synchronization Job
Section 3: Interface Method Descriptions

*Note: The reader is assumed to be familiar with the TrueSync Marketing Requirements and TrueSync Functional Specification documents. Information in these documents is not repeated here, unless the implementation is noteworthy to the developer.*

## *Section 1: Developer's Overview of the TrueSync Architecture*

### Introduction

TrueSync is designed to be a highly configurable and extensible synchronization engine. An architectural framework and several toolkits make it easy for the developer to add support for new datasets or change the behavior of synchronization for existing datasets.

However, although TrueSync if fairly simple and the amount of code is quite modest, the price of this flexibility is a relatively steep learning curve for the developer. This guide attempts to state TrueSync's assumptions, mechanisms and expectations to accelerate the developer's familiarization.

### Dataset Assumptions

TrueSync makes the following assumptions about the entities it can synchronize:

- The entities that TrueSync works with are called ***datasets***. Each dataset is composed of a countable number of ***records***.
- Each record can be uniquely associated with an ***identifier*** throughout its lifetime. Each record also has a ***timestamp*** noting the time of last modification.
- The format of a timestamp and identifier within a dataset are completely arbitrary. However, the timestamp must be transformable to and from a 32-bit signed integer representing UTC time in seconds. The C++ primitive or class that encapsulates the identifier must support a default constructor, assignment and equality operators and be serializable to and from a binary file.

### Synchronization Network

TrueSync is designed to work in an arbitrary synchronization topology. That is you can "sync" whatever you want with anything else, whenever you want. TrueSync utilizes supplemental data files and protection logic to insure synchronization stability.

Key to this stability is the assumption that each dataset and each record within that dataset can be known within the TrueSync network by a globally unique identifier, called the ***global id***. During

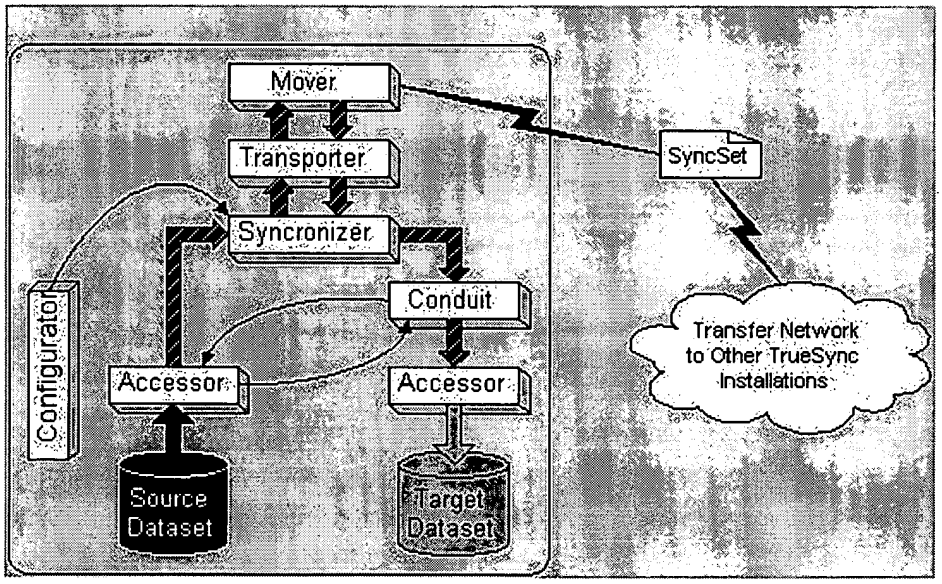
synchronization, dataset specific record identifiers are matched with global ids.

TrueSync provides a C++ template class (`AccessKit\TRecordMap`) for storage of and lookup within the mapping from global id to specific identifier.

## TrueSync Module architecture

TrueSync consists of a collection of dynamic link libraries, that have a "plug-and-play" relationship. Each DLL is explicitly loaded and provides an exported function that serves as a "object factory". TrueSync provides a class to manage the consumer side of this relationship (`ModuleKit\ObjectRequester`).

TrueSync modules are of three types: ***synchronization core***, ***accessor*** and ***conduit***, shown in the following figure: (Ignore the ***transporter*** and ***mover*** modules, they might be implemented in a future email-based TrueSync implementation.)




The default ***synchronization core*** module is called `SyncCore`. It is anticipated that only rarely will `SyncCore` be replaced at runtime with alternate synchronization logic. `SyncCore` provides the synchronization policies and procedures dealing with issues like:

- Has a record been inserted, updated or deleted?
- What happens when a record has been modified in both datasets between synchronization processing?

An ***accessor*** module provides all of the interaction with a specific dataset type, such as Sidekick Cardfiles, or Intera Events. Accessors deal with issues such as:

- Opening and closing a dataset.
- Insertion, update and deletion of record for this dataset type.
- Serialization of records to/from binary disk files (justified below).
- Communication with application "owning" dataset (such as Sidekick).

A ***conduit*** module transforms a record from a source dataset format into a target dataset format that can be inserted into the target dataset or used to update an existing target record.

## Directional Synchronization and Syncsets

An original goal of the TrueSync architecture was to support synchronization by email. Thus it was

assumed that the two datasets comprising a synchronization relationship were not necessarily directly accessible by the same PC. So synchronization was broken down into two directions (***forward*** and ***reverse***) and each direction was broken down into two phases (***outbound*** and ***inbound***).

The two directions are determined by the roles each dataset is playing. That is, who is the ***source*** dataset and who is the ***target*** dataset. The phases are determined by whether we are reading records from a source (outbound), or are writing records to a target (inbound).

During the outbound phase records are read from the source dataset and written to a binary file called a ***syncset***. During the inbound phase records are read from the syncset and written into the target dataset.

Accessor modules are involved in both the outbound and the inbound phases. Accessor modules are usually named to identify them with the dataset type they support, for example, `IntAddress.dll` for the Intera address accessor.

Conduits are only involved in the inbound phase. Conduits are usually named to represent the transformation they support, for example, `IntAddrSIS.dll` for the Intera to Sidekick address conduit.

## TrueSync Configuration

At runtime, TrueSync behavior is controlled by an initialization file. This initialization file stores information about which datasets are to be synchronized, which accessors and conduits to use, and parameters that these accessors require to locate and access their datasets.

Note that any DLL which contains the "object factory" export function and returns properly constructed and functioning requested objects may be specified in the configuration. The same DLL may be used in all four roles (dataset one accessor and conduit, dataset two accessor and conduit). However, this is not the way the current Intera and Sidekick DLLs are organized.

## Exported Objects, Virtual Table Linking and Toolkits

The C++ object that is actually returned from each module's object factory is an instance of an abstract base class (or ***ABC***). A programmer is free to implement this ABC anyway she wants. Standard implementations are provided in the various toolkits and these are used by the existing Intera and Sidekick modules. Use of ABCs in this fashion provides a sort of virtual table linking between TrueSync modules. As long as the exported ABCs do not change, modules may be updated as desired.

In order to support this "plug-and-play" approach to software, no standard utility library is provided with TrueSync. Instead, various ***toolkits*** are provided. Toolkits are collections of C++ source code that are included into conduits and accessors at compile time. This is accomplished by using the preprocessor `#include` directive to include the required `*.cpp` file in each module's `ToolKit.cpp` file.

While this results in some code bloat (TrueSync has a fairly small disk footprint), Toolkit classes may be changed arbitrarily, without worry of the effect on existing installed modules.

For historical reasons, TrueSync is dependent on the MFC class libraries, however.

# TrueSync Developer's Guide

## *Section 2: Lifecycle of a Synchronization Job*

### Introduction

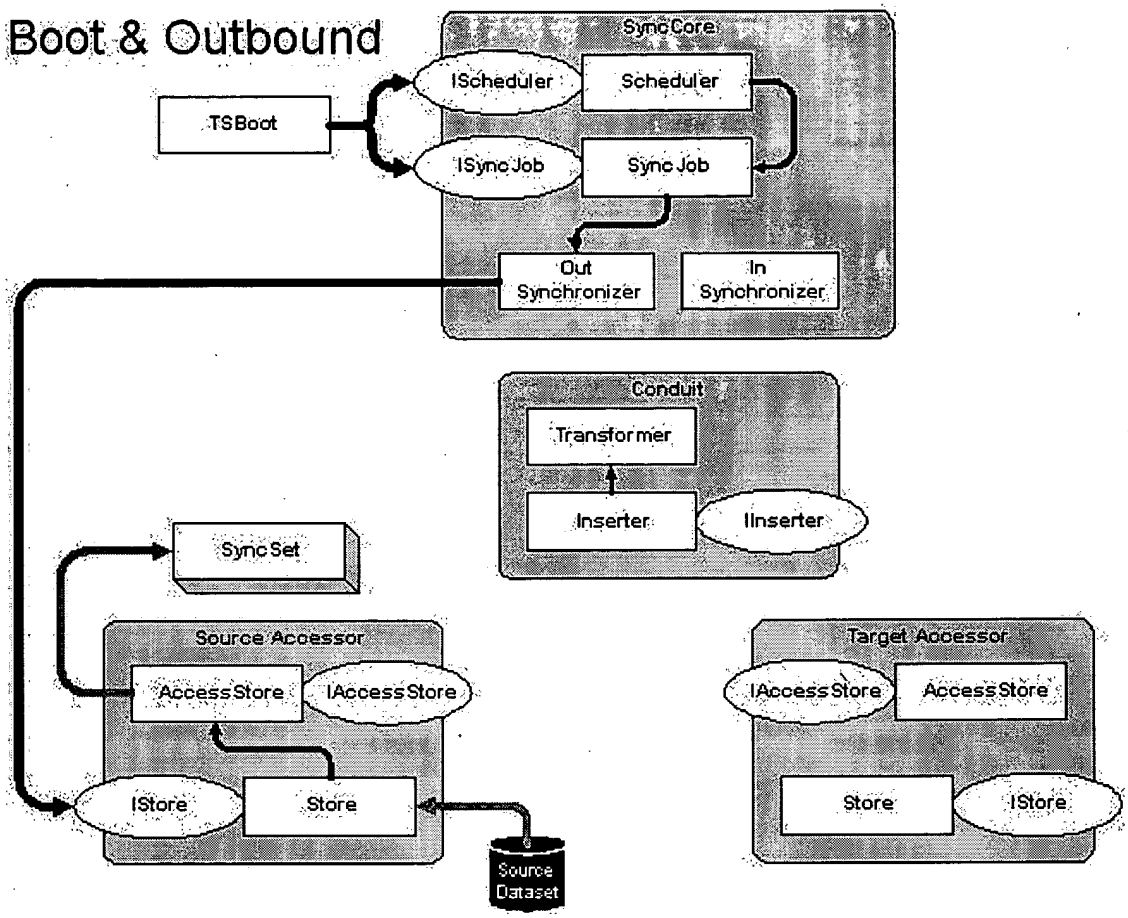
This section describes the sequence of events that comprise the ***boot***, ***outbound*** and ***inbound*** phases.

Note that a ***Dataset*** refers to a persistent entity that is a source or target for records. A ***Store*** is a transient object that is instantiated to access records from a dataset. The unit of synchronization for TrueSync is actually the store and not the dataset. That is, the initialization file refers to stores.

### Boot Phase Events

Refer to the figure below for a diagram of the interfaces and objects in the following events.

1. Client software (such as, a user interface ActiveX control) instantiates the `ModuleKit\TSBoot` class. This class must remain in existance throughout the lifetime of any active synchronization jobs.
2. The `TSBoot` object determines the `SyncCore` module to use and the directory where TrueSync modules reside.
3. `TSBoot` then loads the `SyncCore` module and instantiates the `SyncCore\Scheduler` object.
4. Using the Registry, `TSBoot` locates the TrueSync working directory where the initialization file will be found.
5. `TSBoot` asks the Scheduler to create a `SyncCore\SyncJob` object. `TSBoot` then instructs the `SyncJob` to read the job configuration information from the initialization file.
6. `TSBoot` then instructs the `SyncJob` to perform the synchronization.
7. SyncJob determines which combination of forward and/or reverse synchronization has been requested, puts the datasets into either the source or target role, and performs the outbound and inbound phases.

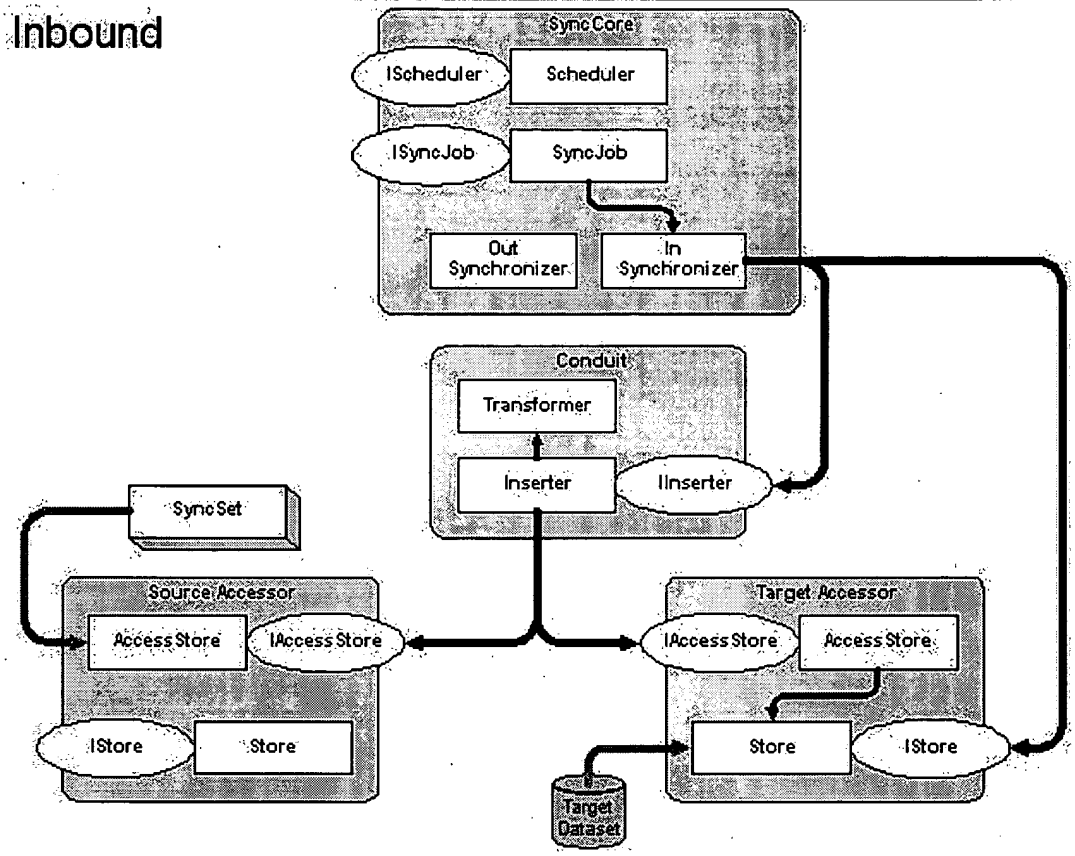



## Outbound Phase Events

Refer to the figure above for a diagram of the interfaces and objects in the following events. Note that the conduit and target accessor modules are not involved during the outbound phase.

1. `SyncJob` creates and opens for writing a `SyncSet` file using the MFC `CFile` class.
2. `SyncJob` then instantiates a `SyncCore\OutSynchronizer` object and calls its **open** method.
3. Using the `ModuleKit\ObjectRequester`, `OutSynchronizer` loads the source accessor, instantiates a `Store` object and calls its **open** method.
4. Using the information extracted from the initialization file, `Store` opens the dataset for reading and opens the dataset's corresponding `AccessKit\RecordMap` (not shown).
5. Control returns to `OutSynchronizer`, who opens the `SyncCore\TransactionTable` (not shown) for this synchronization relationship.
6. Control returns to `SyncJob` which creates a MFC `CArchive` class for writing to the `Syncset` and then instructs the `OutSynchronizer` to perform the actual outbound synchronization.
7. `OutSynchronizer` performs outbound synchronization. The policies and procedures for this are documented in the *TrueSync Functional Specification* document and will not be duplicated here. Other sections discuss the methods that `OutSynchronizer` calls on the `Store` object, which a developer needs to satisfy when programming a new source accessor.
8. `SyncJob` calls the **close** method of the `OutSynchronizer`, which causes the other dependent **close** methods to be called in the reverse order that their **open** methods were called. `SyncJob` closes the `SyncSet`.
9. `SyncJob` formats statistics and via a Windows message and/or a callback function, returns

these statistics to the client.




## Inbound Phase Events

Refer to the figure above for a diagram of the interfaces and objects in the following events. Note that the source accessor module is only used to read the `syncset` and the source `store` is not opened.

1. `SyncJob` creates and opens for reading a `SyncSet` file using the MFC `CFile` class.
2. `SyncJob` then instantiates a `SyncCore\InSynchronizer` object and calls its **open** method.
3. Using the `ModuleKit\ObjectRequester`, `InSynchronizer` loads the conduit, instantiates a `AccessKit\Inserter` object and calls its **open** method.
4. Using its own `ObjectRequester`, `Inserter` loads both accessor modules and requests an `AccessKit\AccessStore` object from each. `Inserter` also requests a Store object from the target `AccessStore`.
5. `Inserter` calls the open method on the target `Store`.
6. Using the information extracted from the initialization file, `Store` opens the dataset for writing and opens the dataset's corresponding `AccessKit\RecordMap` (not shown).
7. Control returns to `Inserter`, who instructs `Transformer` to read any transformation configuration information from the initialization file.
8. Control returns to `InSynchronizer`, who opens the `SyncCore\TransactionTable` (not shown) for this synchronization relationship.
9. Control returns to `SyncJob` which creates a MFC `CArchive` class for reading from the `Syncset` and then instructs the `InSynchronizer` to perform the actual inbound synchronization.
10. `InSynchronizer` performs inbound synchronization. The policies and procedures for this

are documented in the *TrueSync Functional Specification* document and will not be duplicated here. Other sections discuss the methods that `InSynchronizer` calls on the `Store` and `Inserter` objects and the methods that `Inserter` calls on the `AccessStore` objects, which a developer needs to satisfy when programming a new conduit and target accessor.

11. `SyncJob` calls the **close** method of the `InSynchronizer`, which causes the other dependent **close** methods to be called in the reverse order that their **open** methods were called. `SyncJob` closes and deletes (if the job was successful) the `SyncSet`.
12. `SyncJob` formats statistics and via a Windows message and/or a callback function, returns these statistics to the client.

# TrueSync Developer's Guide

## *Section 3: Interface Method Descriptions*

### Introduction

This section of the Developers Guide discusses in detail the interface methods that a programmer needs to satisfy when implementing an accessor or conduit.

Note that the `SyncCore.dll` implementation of the `SyncCore` logic only requires the use of the `SyncCore\IStore` and `ConduitKit\IInserter` interfaces. The `AccessKit\IAccessStore` interface used inbound between the conduit and the accessors is optional and assumes the use of data storage classes from `ToolKit`. This is also true of the implementation base class `ConduitKit\Transformer`.

Note that during the outbound phase, the `Store` is the one and only interface object that is instantiated.

### Interface List


- IStore *(outbound & inbound)*
- IStore *(outbound only)*
- IStore *(inbound only)*
- IAccessStore *(source & target)*
- IAccessStore *(source only)*
- IAccessStore *(target only)*


---

### IStore *(outbound & inbound)*

**`void destroy()`**

A request to delete the object. Usually implemented inline as `{ delete this; }`.

**`int open(ISyncInfo& info, int mode)`**
**`int close()`**

`Open(...)` is a request to open the dataset and the record map. Configuration information may be retrieved from `ISyncInfo`. The dataset mode is `READ` for outbound processing and `WRITE` for inbound processing.

`Close()` is a request to close the dataset and the record map.

Return non-zero for success and zero for failure.

**`IRecordMap& map()`**

A request for the record map. Your store must instantiate a class using the `AccessKit\TRecordMap` template. See the header file for details.

**`int moveRecordToFromMap()`**

An instruction to locate in the dataset the record that corresponds to the "current" record in the `RecordMap`. Called outbound during delete order generation and inbound for both deletes and exports. The current record in the `RecordMap` is retrieved with the method `MapEntryExt& TRecordMap::current()`.

Return non-zero for success and zero for failure. If the record is located, it should be made the "current" record in the `Store`. This is usually accomplished by keeping a copy of the record in memory.

**`time_t recordModifyTime()`**

A query for the `Store`'s "current" record last modified timestamp.

---

IStore *(outbound only)*

**`int extractByDate(time_t t)`**

A request to prepare for outbound export record generation. The parameter is the time of the last outbound synchronization with the target dataset. This method allows large datasets to limit the number of records extracted for processing. After this call the extracted recordset is assumed ready for iteration. Return non-zero for success and zero for failure.

**`int isEmpty()`**
**`int isEOF ()`**

Queries about the state of extraction iteration. Called during export generation. `isEmpty()` should return non-zero if there are records to iterate over. `isEOF()` should return non-zero if iteration has traversed all records in the extracted set. The `OutSynchronizer` logic does not require multiple or reverse iteration of the extract set.

**`void moveRecordFirst()`**
**`void moveRecordNext ()`**

Instructions to iterate through the extracted records. Called during export generation. The record moved to becomes the "current" record in the `Store`.

**`MapEntryExt* moveEntryToFromStore()`**

An instruction to locate in the `RecordMap` the entry that corresponds to the "current" record in the `Store`. Called during export generation. Use the call `TRecordMap::moveToInt(MAP_ENTRY_INT&)`. Return the located entry or `NULL` if not found.

**`MapEntryExt& addToMapForExport()`**

An instruction to add an entry into the `RecordMap` containing the `Store`'s current record internal identifier and timestamp. Called during export generation. Use the call `TRecordMap::addExport(MAP_ENTRY_INT&, const CTime&)`. Return the new map entry.

**`int filterForExport()`**

The `store` may perform filtering to block certain records from being exported. Called during export generation. Return non-zero to pass the record and zero to block it.

**`void sendSchema(CArchive& ar)`**

A request to serialize the `Store`'s schema into the passed MFC archive. This request may be ignored if the schema is fixed, like Intera. Called during export generation, before record serialization.

The format is completely at the discretion of the `Store`, although the `ToolKit\FieldDirectory` class is available for describing an arbitrary schema.

**`void sendRecord(CArchive& ar)`**

A request to serialize the `Store`'s "current" record into the passed MFC archive. The format is completely at the discretion of the `Store`, although the `ToolKit\FieldDirectory` class is available for describing an arbitrary data structure. Called during export generation.

Return to Interface List

---

## IStore Interface *(inbound only)*

**`MapEntryExt& addToMapForImport(MapEntryExt& entryExt, time_t timestamp)`**

An instruction to add an entry into the `RecordMap` containing the `Store`'s current record internal identifier and the passed `MapEntryExt` and `timestamp`. Called after a record is inserted into the Store. Use the call `TRecordMap::addImport(MAP_ENTRY_INT&, MapEntryExt&, time_t)`. Return the new map entry.

**`void removeRecord()`**

Instruction to remove the `Store` processing.

Return to

---

IAccessStore Interface *(source & target)*

A request to delete the object. Usually implemented inline as `{ delete this; }`

---

*(source only)*

**ITransHalf& recvSchema(CArchive& ar)**

`Store`'s schema out of the passed MFC archive. The serialization request may be ignored if the schema was not serialized outbound. Called if the schema is found while `SyncSet`.

`ToolKit\TransHalf` object describing the schema. A consists of a last-modified timestamp and two , one for the schema and one for the "current" `TransHalf` and header files.

If you want to use the interface you need to become familiar with the building blocks of , namely `Fields` `FieldArrays`.

A request to create or overwrite the `Store` passed MFC archive. Called if a record is found while reading the .

Return to

---

IAccessStore Interface *(target only)*

**ITransHalf& getSchema()**

**void fetchTgtFields()**

**void acceptRecord(const ITransHalf& th)**

**int filterForDelete()**

**int filterForInsert()**

**time_t insertRecord()**

**time_t updateRecord()**